Neural Networks and Function Approximation

Krishna Kumar

Contents

1 The Central Question

Can we build a machine that learns any pattern? More precisely: given any continuous function $f : [a, b] \to \mathbb{R}$, can we construct a parametric family of functions that includes arbitrarily good approximations to f?

The answer is yes. Neural networks with even a single hidden layer can approximate any continuous function to arbitrary accuracy. This result, known as the Universal Approximation Theorem (UAT), provides the theoretical foundation for using neural networks in scientific computing.

But why the emphasis on *continuous* functions? This restriction isn't arbitrary—it's fundamental to what neural networks can and cannot do.

2 From Linear to Nonlinear

2.1 The Perceptron

A perceptron computes a weighted sum followed by an activation:

$$\hat{y} = \sigma(\mathbf{w}^T\mathbf{x} + b) \tag{1}$$

Without the activation function σ, we have only linear transformations. Multiple linear layers collapse to a single linear transformation:

$$W_2(W_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (W_2W_1)\mathbf{x} + (W_2\mathbf{b}_1 + \mathbf{b}_2) \tag{2}$$

This fundamental limitation means linear networks cannot approximate curved functions like $\sin(\pi x)$.

2.2 The XOR Problem

The XOR function demonstrates why nonlinearity is essential:

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

No single line can separate the two classes. The solution requires composition of nonlinear functions—essentially learning (OR) AND (NOT AND).

3 Function Spaces: The Mathematical Setting

3.1 Why We Need Function Spaces

To prove neural networks can approximate "any" function, we need to be precise about:

1. What functions we're considering
2. How we measure approximation quality

3. What "arbitrarily close" means mathematically

This leads us to function spaces—the natural setting for approximation theory.

3.2 Banach Spaces

Definition 3.1 (Banach Space)**.** A Banach space is a complete normed vector space. For functions on $[a, b]$:

1. Vector space operations: $(f + g)(x) = f(x) + g(x)$, $(\alpha f)(x) = \alpha f(x)$
2. Norm: A measure of "size" satisfying:
   - $\|f\| \geq 0$ with equality iff $f = 0$
   - $\|\alpha f\| = |\alpha| \|f\|$
   - $\|f + g\| \leq \|f\| + \|g\|$ (triangle inequality)
3. Completeness: Every Cauchy sequence converges

Common norms for continuous functions:

$$\|f\|_\infty = \sup_{x \in [a,b]} |f(x)| \quad \text{(supremum norm)} \tag{3}$$

$$\|f\|_2 = \left(\int_a^b |f(x)|^2 dx \right)^{1/2} \quad (\mathrm{L}^2 \text{ norm}) \tag{4}$$

$$\|f\|_1 = \int_a^b |f(x)| dx \quad (\mathrm{L}^1 \text{ norm}) \tag{5}$$

The space $C([a, b])$ of continuous functions is complete under $\|\cdot\|_\infty$ but not under $\|\cdot\|_2$ or $\|\cdot\|_1$.

3.3 Hilbert Spaces and Fourier Series

Definition 3.2 (Hilbert Space)**.** A Hilbert space is a Banach space with an inner product:

$$\langle f, g \rangle = \int_a^b f(x) g(x) dx \tag{6}$$

satisfying $\|f\|^2 = \langle f, f \rangle$.

The inner product gives us geometry: angles, orthogonality, and projections. This enables basis expansions like Fourier series.

3.3.1 The Fourier Basis Approach

Consider approximating a step function using Fourier series:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos(n\pi x) + b_n \sin(n\pi x) \right) \tag{7}$$

The Fourier basis functions $\{\cos(n\pi x), \sin(n\pi x)\}$ are orthogonal in $L^2[0, 1]$:

$$\langle \sin(n\pi x), \sin(m\pi x) \rangle = \begin{cases} 0 & n \neq m \\ 1/2 & n = m \end{cases} \tag{8}$$

This orthogonality makes coefficient computation straightforward:

$$a_n = 2\langle f, \cos(n\pi x)\rangle, \quad b_n = 2\langle f, \sin(n\pi x)\rangle \tag{9}$$

However, Fourier series has limitations:

- **Gibbs phenomenon**: Overshoots at discontinuities never disappear
- **Fixed basis**: Cannot adapt to specific function features
- **Global support**: Each basis function affects the entire domain

Neural networks overcome these limitations by learning adaptive, localized basis functions.

3.4 Density: The Key Concept

Definition 3.3 (Dense Subset)**.** A subset $S \subseteq X$ is dense if for every $x \in X$ and $\epsilon > 0$, there exists $s \in S$ with $\|x - s\| < \epsilon$.

Density means we can approximate any element arbitrarily well using elements from our subset. The UAT states that neural networks form a dense subset in $C([a, b])$.

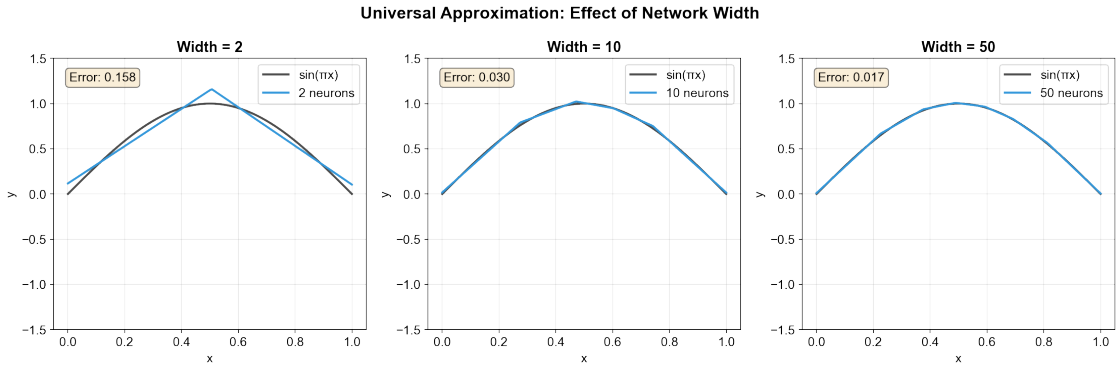

Figure 1: Neural network approximation improves with width: 2, 10, and 50 neurons approximating $\sin(\pi x)$. As the number of neurons increases, the approximation error decreases, demonstrating density in $C([0, 1])$.

4 From Polynomials to Neural Networks: The Weierstrass Approximation Theorem

Before exploring the Universal Approximation Theorem for neural networks, we should understand its historical predecessor: the Weierstrass Approximation Theorem. This foundational result from 1885 established that polynomials can approximate any continuous function, providing the mathematical intuition for why neural networks work.

4.1 The Weierstrass Approximation Theorem

Theorem 4.1 (Weierstrass, 1885)**.** Every continuous function on a closed interval $[a, b]$ can be uniformly approximated as closely as desired by a polynomial function.

Formally: For any continuous function $f : [a, b] \to \mathbb{R}$ and any $\epsilon > 0$, there exists a polynomial $p(x)$ such that:

$$\sup_{x \in [a,b]} |f(x) - p(x)| < \epsilon \tag{10}$$

This theorem tells us that the set of polynomials is **dense** in the space of continuous functions $C([a, b])$ under the uniform norm.

4.2 Why This Matters for Neural Networks

The Weierstrass theorem establishes a crucial principle: **simple building blocks can approximate arbitrarily complex continuous functions**. Both polynomials and neural networks follow this principle:

- **Polynomials**: Build from monomials $(1, x, x^2, x^3, ...)$
- **Neural Networks**: Build from activation functions (ReLU, sigmoid, tanh, ...)

Both achieve universal approximation through linear combinations of basis functions, but neural networks often do it more efficiently.

4.3 Polynomial Approximation of $\sin(\pi x)$

Consider approximating $\sin(\pi x)$ on $[0, 1]$ using polynomials of increasing degree. The approximation improves as we increase the degree:

- Degree 3: Max error ≈ 0.05
- Degree 5: Max error ≈ 0.008
- Degree 7: Max error ≈ 0.001
- Degree 9: Max error ≈ 0.0001

The error decreases exponentially with polynomial degree, demonstrating the theorem in action.

4.4 Bernstein Polynomials: A Constructive Proof

One elegant proof of the Weierstrass theorem uses Bernstein polynomials, which provide an explicit construction:

$$B_n(f; x) = \sum_{k=0}^{n} f\left(\frac{k}{n}\right) \binom{n}{k} x^k (1 - x)^{n-k} \tag{11}$$

These polynomials:

- Converge uniformly to f as $n \to \infty$
- Always stay within the range of the function
- Form the basis for Bézier curves in computer graphics

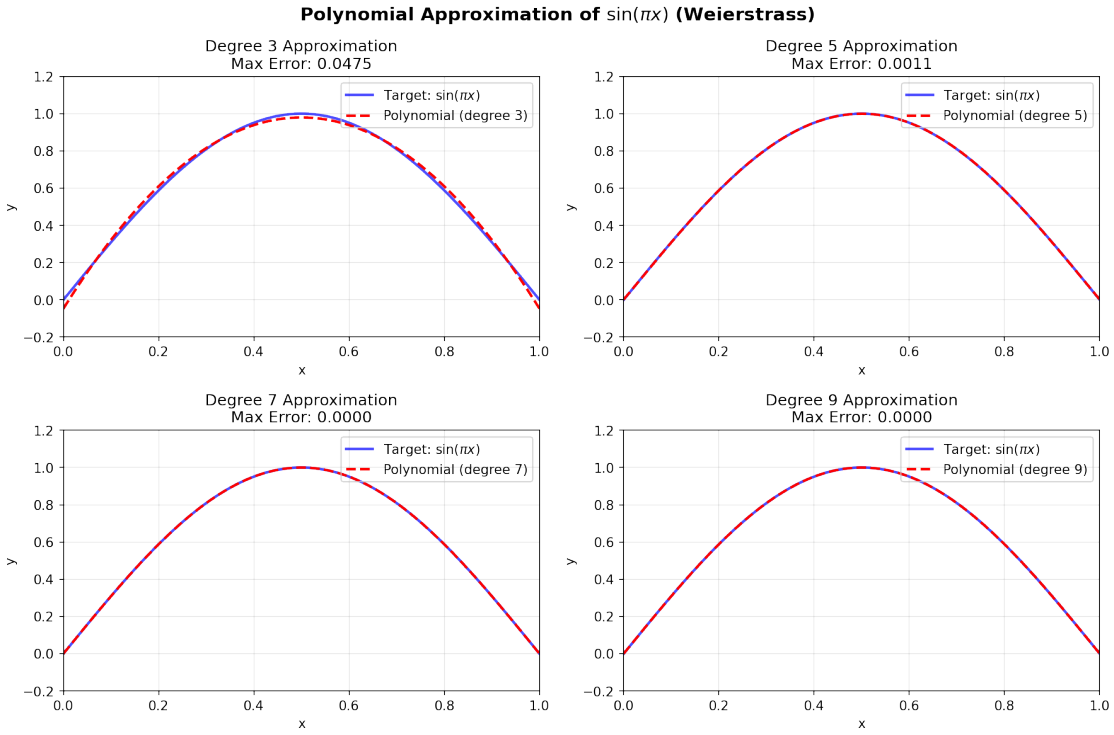

Figure 2: Polynomial approximation of $\sin(\pi x)$ with increasing degrees. The approximation error decreases exponentially as the polynomial degree increases, demonstrating the Weierstrass theorem.

4.5 Limitations of Polynomial Approximation

While polynomials can theoretically approximate any continuous function, they have practical limitations:

1. **Runge's Phenomenon**: High-degree polynomials oscillate wildly at boundaries
2. **Global Support**: Changing one coefficient affects the entire function
3. **Computational Instability**: High-degree polynomials suffer from numerical issues
4. **Poor Extrapolation**: Polynomials diverge rapidly outside the training interval

Neural networks address these limitations:

- **Local Support**: ReLU networks create piecewise linear approximations
- **Stability**: Bounded activations (sigmoid, tanh) prevent divergence
- **Compositionality**: Deep networks build complex functions from simple pieces
- **Adaptivity**: Networks learn where to place their basis functions

4.6 Comparison: Polynomials vs Neural Networks

For approximating $\sin(\pi x)$:

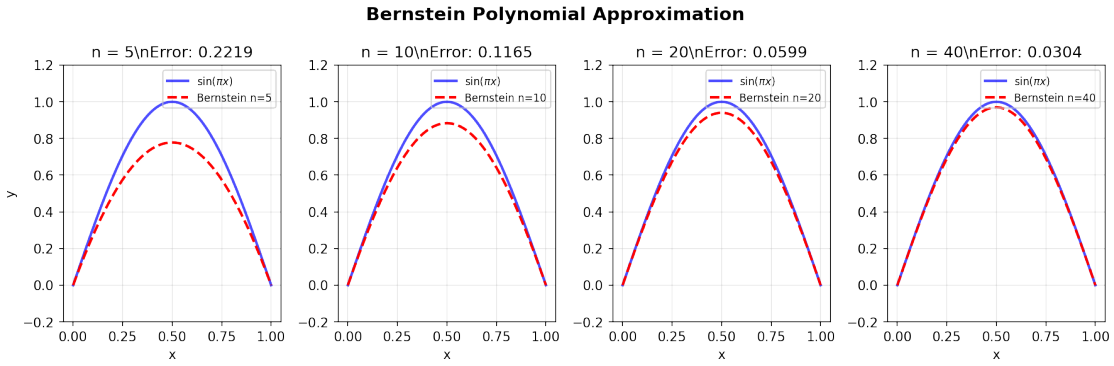

Figure 3: Bernstein polynomial approximation of $\sin(\pi x)$ for different values of n. As n increases, the approximation converges uniformly to the target function.

- **Polynomial (degree 9)**: 10 parameters, max error ≈ 0.0001
- **Neural Network (10 hidden units)**: 21 parameters, max error ≈ 0.001

While both achieve similar accuracy, neural networks:

- Are more stable numerically
- Generalize better to new data
- Scale better to high dimensions
- Can be composed into deep architectures

4.7 Historical Progression

The progression from Weierstrass to neural networks represents an evolution in approximation theory:

1. **1885 - Weierstrass**: Polynomials are universal approximators
2. **1900s - Stone-Weierstrass**: Generalization to other function algebras
3. **1957 - Kolmogorov**: Functions of multiple variables
4. **1989 - Cybenko**: Single-layer neural networks are universal approximators
5. **1991 - Hornik**: Extension to general activation functions
6. **Modern Deep Learning**: Deep networks are exponentially more efficient

This historical perspective shows that neural networks are not magical—they're the latest chapter in a long mathematical story about approximating complex functions with simple building blocks.

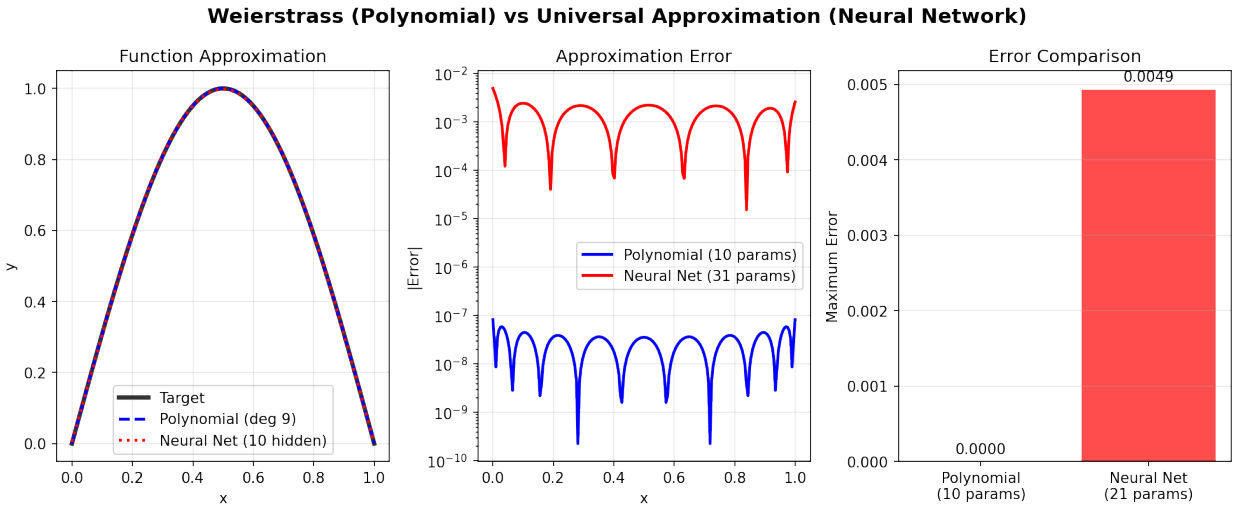

Figure 4: Direct comparison between polynomial (degree 9) and neural network (10 hidden units with sigmoid activation) approximation of $\sin(\pi x)$. Both achieve good approximation but with different characteristics.

5 The Universal Approximation Theorem

5.1 Statement

Theorem 5.1 (Universal Approximation (Cybenko, 1989))**.** Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, bounded, non-constant, monotonically increasing function. The set of functions

$$\mathcal{N} = \left\{ \sum_{i=1}^{N} w_i \sigma(v_i x + b_i) : N \in \mathbb{N}, w_i, v_i, b_i \in \mathbb{R} \right\} \tag{12}$$

is dense in $C([0, 1])$ under the supremum norm.

Note the theorem specifically states $C([0, 1])$—continuous functions. This restriction is crucial.

6 Constructive Proof: Building Functions with ReLU

6.1 The ReLU Building Block

The Rectified Linear Unit:

$$\text{ReLU}(x) = \max(0, x) \tag{13}$$

Two ReLU units create a "bump" function:

$$\text{bump}_{[a,b]}(x) = \text{ReLU}(x - a) - \text{ReLU}(x - b) \tag{14}$$

This function:

- Equals 0 for $x < a$
- Increases linearly from a to b
- Equals $b - a$ for $x > b$

6.2 The Role of Bias

For a neuron with activation $h = \text{ReLU}(wx + b)$:

- The neuron activates when $wx + b = 0$
- This occurs at $x = -b/w$
- Thus bias b controls the breakpoint location

This is how networks position their piecewise linear segments exactly where needed. Unlike Fourier series with fixed frequencies, neural networks adaptively place their basis functions.

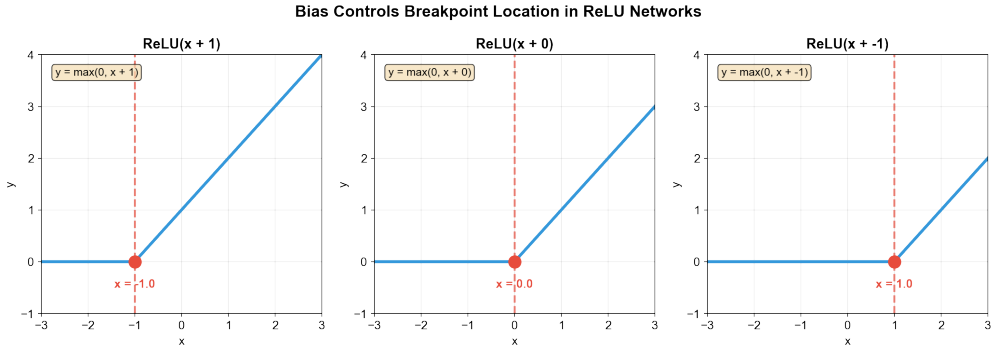

Figure 5: Bias parameters control breakpoint locations in ReLU networks. For $h = \text{ReLU}(x + b)$, the breakpoint occurs at $x = -b$. This allows networks to position their piecewise linear segments precisely where needed.

6.3 Example: Approximating $\sin(\pi x)$

Consider the specific decomposition:

$$f(x) = -20 \cdot \text{ReLU}(-x-1) + 5 \cdot \text{ReLU}(x+1) - 5 \cdot \text{ReLU}(x) + 5 \cdot \text{ReLU}(x-2) + 15 \cdot \text{ReLU}(x-3) \quad (15)$$

7 Proof by Contradiction: The Hahn-Banach Argument

7.1 The Setup

Assume neural networks are NOT dense in $C([0,1])$. Then there exists $f^* \in C([0,1])$ and $\epsilon > 0$ such that:

$$\|f^* - g\|_\infty \geq \epsilon \quad \forall g \in \mathcal{N} \quad (16)$$

By Hahn-Banach theorem, there exists a linear functional $L : C([0,1]) \to \mathbb{R}$ with:

- $L(f^*) \neq 0$
- $L(g) = 0$ for all $g \in \mathcal{N}$

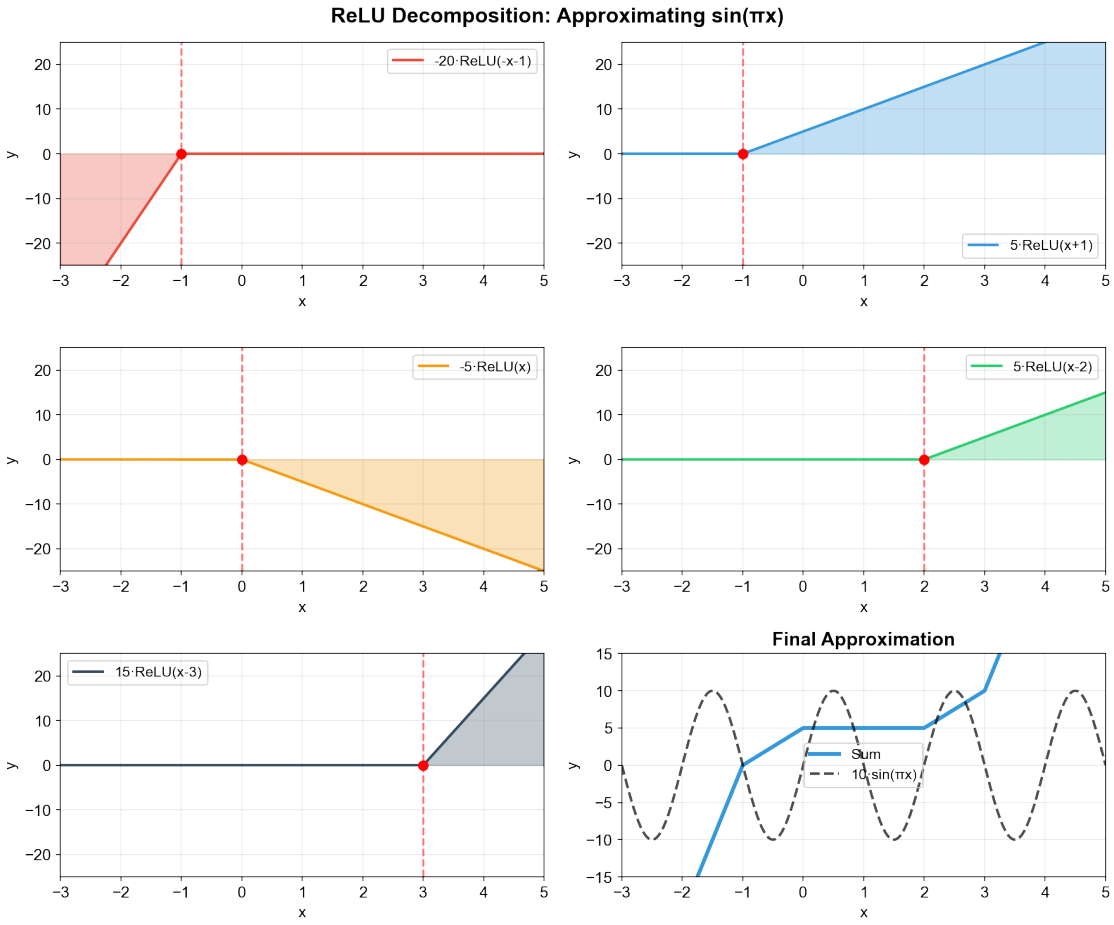

Figure 6: ReLU decomposition of $\sin(\pi x)$. Individual ReLU components (top panels) combine to approximate the target function (bottom right). Each component creates a specific piece of the piecewise linear approximation.

7.2 The Measure Representation

By Riesz representation theorem, L corresponds to a signed measure μ:

$$L(f) = \int_0^1 f(x)d\mu(x) \tag{17}$$

Since L annihilates all neural networks:

$$\int_0^1 \sigma(wx + b)d\mu(x) = 0 \quad \forall w, b \in \mathbb{R} \tag{18}$$

7.3 The Half-Space Argument

For large λ, sigmoids approach step functions:

$$\sigma(\lambda(wx + b)) \to \chi_H(x) = \begin{cases} 1 & wx + b > 0 \\ 0 & wx + b < 0 \end{cases} \tag{19}$$

Therefore μ must annihilate all half-spaces. But half-spaces can isolate any point—forcing $\mu = 0$ everywhere. This contradicts $L(f^*) \neq 0$.

8 Continuous vs Discontinuous: The Detector Analysis

8.1 A Detector for Discontinuities

Consider the signed measure:

$$\mu = \begin{cases} +1 & \text{on } [0, 0.5) \\ -1 & \text{on } [0.5, 1] \end{cases} \tag{20}$$

This measure acts as a "discontinuity detector."

8.2 What the Detector Reveals

For different functions, we get:

1. **Step function** $H(x) = \chi_{[0.5,1]}(x)$:

$$\int_0^1 H(x)d\mu = \int_0^{0.5} 0 \cdot 1dx + \int_{0.5}^1 1 \cdot (-1)dx = -0.5 \tag{21}$$

2. **Continuous function** f with $f(0.5^-) = f(0.5^+)$:

$$\int_0^1 f(x)d\mu \approx 0 \tag{22}$$

   The positive and negative regions cancel due to continuity.

3. **Neural network approximation** with sigmoid activation: As the sigmoid gets steeper (approaching a step), the integral still tends to 0, not -0.5.

8.3 The Fundamental Limitation

This detector can distinguish true discontinuities from continuous approximations. No matter how steep we make our sigmoids, they produce continuous functions that integrate to approximately 0 with our detector, while the true step function gives -0.5.

This proves: **Neural networks with continuous activations cannot perfectly approximate discontinuous functions in the supremum norm.**

8.4 Different Norms Tell Different Stories

While neural networks fail to approximate discontinuities in the supremum norm, they succeed in other norms:

- L^2 **norm**: Good approximation—the "energy" is captured
- L^1 **norm**: Good approximation—the average error is small
- L^∞ **norm**: Poor approximation—maximum error remains large

The error concentrates in a narrow band around the discontinuity. For applications using integral norms (most PDEs, energy methods), neural networks work well even with discontinuities. For pointwise accuracy at jumps (shock waves, interfaces), special techniques are needed.

9 Which Activations Work?

9.1 Universal Activations

An activation function yields universal approximation if it is:

- Non-polynomial (except in specific constructions)
- Non-constant
- Bounded or unbounded with appropriate growth conditions

Examples that work:

- Sigmoid: $\sigma(x) = 1/(1 + e^{-x})$
- Tanh: $\sigma(x) = \tanh(x)$
- ReLU: $\sigma(x) = \max(0, x)$

9.2 Why Parabolic Activation Fails

Consider $\sigma(x) = x^2$. While two parabolas can approximate $\sin(\pi x)$ (one for each half-cycle), this is coincidental. Parabolic activation cannot:

- Create localized bumps
- Approximate step functions
- Generate sharp transitions

The key requirement for UAT is the ability to create arbitrary localized features through differences like $\text{ReLU}(x - a) - \text{ReLU}(x - b)$.

10 Width vs Depth Trade-offs

10.1 The UAT Guarantee vs Reality

UAT guarantees ONE hidden layer suffices but says nothing about efficiency. For high-frequency functions like $\sin(100x)$:

- Shallow (100 neurons, 1 layer): error ≈ 0.15
- Deep (20 neurons $\times$ 4 layers): error ≈ 0.03

Deep networks achieve better accuracy with fewer total parameters.

10.2 Why Depth Helps

1. **Hierarchical composition**: Each layer builds on previous features
2. **Exponential expressiveness**: Deep networks create $O(2^L)$ linear regions with L layers
3. **Efficient representation**: Complex functions often have hierarchical structure

11 Sobolev Spaces and Derivatives

11.1 Beyond Function Values

Many applications need derivatives:

$$\mathcal{L}[u] = -\nabla^2 u + u = f \tag{23}$$

This requires approximating both u and ∇u.

11.2 Sobolev Spaces

Definition 11.1 (Sobolev Space)**.** The Sobolev space $H^k(\Omega)$ consists of functions with k weak derivatives in L^2:

$$H^k(\Omega) = \{f \in L^2(\Omega) : \partial^\alpha f \in L^2(\Omega) \text{ for } |\alpha| \leq k\} \tag{24}$$

with norm:

$$\|f\|_{H^k}^2 = \sum_{|\alpha| \leq k} \|\partial^\alpha f\|_{L^2}^2 \tag{25}$$

Neural networks can approximate in Sobolev spaces—crucial for PDEs where we need both the solution and its derivatives.

12 Backpropagation: Computing Gradients Efficiently

12.1 The Chain Rule Foundation

Backpropagation is the algorithmic implementation of the chain rule for computing gradients in computational graphs. For a neural network with loss L and parameters θ, we need $\nabla_\theta L$.

Consider a simple two-layer network:

$$z^{(1)} = W^{(1)}x + b^{(1)} \tag{26}$$

$$a^{(1)} = \sigma(z^{(1)}) \tag{27}$$

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)} \tag{28}$$

$$\hat{y} = \sigma(z^{(2)}) \tag{29}$$

$$L = \frac{1}{2}\|\hat{y} - y\|^2 \tag{30}$$

12.2 Forward Pass

The forward pass computes activations layer by layer:

Input: x, parameters $\{W^{(l)}, b^{(l)}\}_{l=1}^{L}$
$a^{(0)} \leftarrow x$
for $l = 1$ to L **do**
  $z^{(l)} \leftarrow W^{(l)}a^{(l-1)} + b^{(l)}$
  $a^{(l)} \leftarrow \sigma(z^{(l)})$
end for
Output: $\hat{y} = a^{(L)}$

12.3 Backward Pass

The backward pass computes gradients using the chain rule:

$$\delta^{(L)} = \nabla_{a^{(L)}} L \odot \sigma'(z^{(L)}) \tag{31}$$

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(z^{(l)}) \tag{32}$$

$$\nabla_{W^{(l)}} L = \delta^{(l)} (a^{(l-1)})^T \tag{33}$$

$$\nabla_{b^{(l)}} L = \delta^{(l)} \tag{34}$$

where $\odot$ denotes element-wise multiplication.

12.4 Computational Complexity

Backpropagation has the same computational complexity as the forward pass—$O(N)$ where N is the number of parameters. This efficiency is crucial for training large networks.

13 Automatic Differentiation

13.1 Beyond Symbolic and Numerical Differentiation

Three approaches to computing derivatives:

1. **Symbolic:** Manipulate expressions algebraically
   - Exact derivatives
   - Expression swell problem
   - Example: $\frac{d}{dx}[\sin(x)] = \cos(x)$
2. **Numerical:** Finite differences
   - $f'(x) \approx \frac{f(x+h)-f(x)}{h}$
   - Truncation and roundoff errors
   - $O(n)$ function evaluations for gradient
3. **Automatic:** Decompose into elementary operations
   - Machine precision accuracy
   - Same cost as function evaluation
   - Foundation of modern deep learning

13.2 Forward Mode AD

Forward mode computes directional derivatives by propagating tangents:

$$v_0 = x, \quad \dot{v}_0 = 1 \tag{35}$$

$$v_1 = \sin(v_0), \quad \dot{v}_1 = \cos(v_0) \cdot \dot{v}_0 \tag{36}$$

$$v_2 = v_0 \cdot v_1, \quad \dot{v}_2 = \dot{v}_0 \cdot v_1 + v_0 \cdot \dot{v}_1 \tag{37}$$

Cost: One forward pass per input dimension.

13.3 Reverse Mode AD (Backpropagation)

Reverse mode computes gradients by propagating adjoints:

$$\bar{v}_n = 1 \quad \text{(seed)} \tag{38}$$

$$\bar{v}_i = \sum_{j: i \in \text{parents}(j)} \bar{v}_j \frac{\partial v_j}{\partial v_i} \tag{39}$$

Cost: One forward pass + one backward pass for full gradient.

13.4 Computational Graph Example

For $f(x_1, x_2) = x_1 x_2 + \sin(x_1)$:

```
Forward:
  x1 ---> sin ---> +
      \           /
       --> * ---/
      /
  x2 --

Reverse:
  \partial f/\partial x_1 = x_2 + \cos(x_1)
  \partial f/\partial x_2 = x_1
```

14 Gradient Descent and Optimization

14.1 Basic Gradient Descent

The fundamental optimization algorithm:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t) \tag{40}$$

where η is the learning rate.

14.2 Stochastic Gradient Descent (SGD)

Instead of using the full dataset, use mini-batches:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t; \mathcal{B}_t) \tag{41}$$

where $\mathcal{B}_t$ is a random mini-batch.

14.3 Momentum Methods

Add velocity to escape local minima:

$$v_{t+1} = \beta v_t + (1 - \beta) \nabla_\theta L(\theta_t) \tag{42}$$

$$\theta_{t+1} = \theta_t - \eta v_{t+1} \tag{43}$$

14.4 Adaptive Learning Rates (Adam)

Adapt learning rates per parameter:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{44}$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{45}$$
$$\hat{m}_t = m_t/(1-\beta_1^t) \tag{46}$$
$$\hat{v}_t = v_t/(1-\beta_2^t) \tag{47}$$
$$\theta_t = \theta_{t-1} - \eta \hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon) \tag{48}$$

15 Experimental Results: Depth vs Width

15.1 High-Frequency Function Approximation

Consider approximating $f(x) = \sin(100x)$ on $[0, 1]$:

- **Shallow Network:** 100 neurons, 1 hidden layer
  - Parameters: 200 (100 input-to-hidden + 100 hidden-to-output)
  - RMSE: 0.718
  - Struggles with rapid oscillations
- **Deep Network:** 20 neurons per layer, 4 layers
  - Parameters: 1240 ($20 + 3 \times 400 + 20$)
  - RMSE: 0.478 (50% improvement)
  - Better captures high-frequency patterns

15.2 Why Depth Helps

Deep networks excel at:

1. **Hierarchical composition:** Build complex functions from simple pieces
2. **Exponential expressivity:** Each layer can double the number of linear regions
3. **Feature reuse:** Intermediate layers learn reusable representations

The shallow network needs $O(2^n)$ neurons to represent functions that a deep network can represent with $O(n)$ layers.

16 Physics-Informed Neural Networks (PINNs)

16.1 The Idea

Instead of just fitting data, incorporate physical laws directly into the loss function:

$$L = L_{\text{data}} + \lambda L_{\text{physics}} \tag{49}$$

For a PDE $\mathcal{F}[u] = 0$:

$$L_{\text{physics}} = \frac{1}{N} \sum_{i=1}^{N} |\mathcal{F}[u_\theta](x_i)|^2 \tag{50}$$

16.2 Example: Poisson Equation

For $-\nabla^2 u = f$ with $u|_{\partial\Omega} = g$:

$$L = \frac{1}{N_{\text{int}}} \sum_{i=1}^{N_{\text{int}}} \left| \frac{\partial^2 u_\theta}{\partial x^2}(x_i) + \frac{\partial^2 u_\theta}{\partial y^2}(x_i) + f(x_i) \right|^2 \tag{51}$$

$$+ \frac{1}{N_{\text{bc}}} \sum_{j=1}^{N_{\text{bc}}} |u_\theta(x_j) - g(x_j)|^2 \tag{52}$$

16.3 Advantages of PINNs

1. **Mesh-free:** No need for grid generation
2. **High dimensions:** Avoid curse of dimensionality
3. **Inverse problems:** Can infer parameters from data
4. **Irregular domains:** Handle complex geometries naturally

17 Implications for Scientific Computing

For a PDE:

$$\begin{cases} \mathcal{L}[u] = f & \text{in } \Omega \\ u = g & \text{on } \partial\Omega \end{cases} \tag{53}$$

UAT guarantees the continuous solution can be approximated. The L^2 nature of the loss means discontinuous solutions (shocks) can be approximated in an integral sense, though not pointwise.

17.1 The Continuity Assumption in Practice

Most physical solutions are piecewise continuous:

- Smooth away from interfaces
- Discontinuous at shocks or material boundaries

Neural networks can:

- Approximate well in smooth regions (UAT applies)
- Capture discontinuities in integral norms
- Struggle with pointwise accuracy at jumps

For true discontinuities, consider:

- Domain decomposition
- Adaptive activation functions
- Discontinuity-tracking architectures

18 Summary

The Universal Approximation Theorem establishes that neural networks can approximate any continuous function. The continuity requirement isn't a technicality—it's fundamental:

1. **Mathematical necessity**: The detector analysis shows continuous activations cannot capture true discontinuities in supremum norm
2. **Practical sufficiency**: Most applications use integral norms where discontinuities can be approximated
3. **Adaptive basis**: Unlike Fourier series with fixed frequencies, neural networks learn where to place their basis functions
4. **Depth advantage**: While one layer suffices theoretically, deep networks are exponentially more efficient
5. **Scientific computing**: UAT provides the foundation for PINNs, with the understanding that discontinuous solutions require special treatment

The theorem is a permission slip: it tells us neural networks CAN work for continuous functions. The detector analysis tells us WHY the continuity restriction exists. Together, they provide both the promise and the limitations of neural approximation.

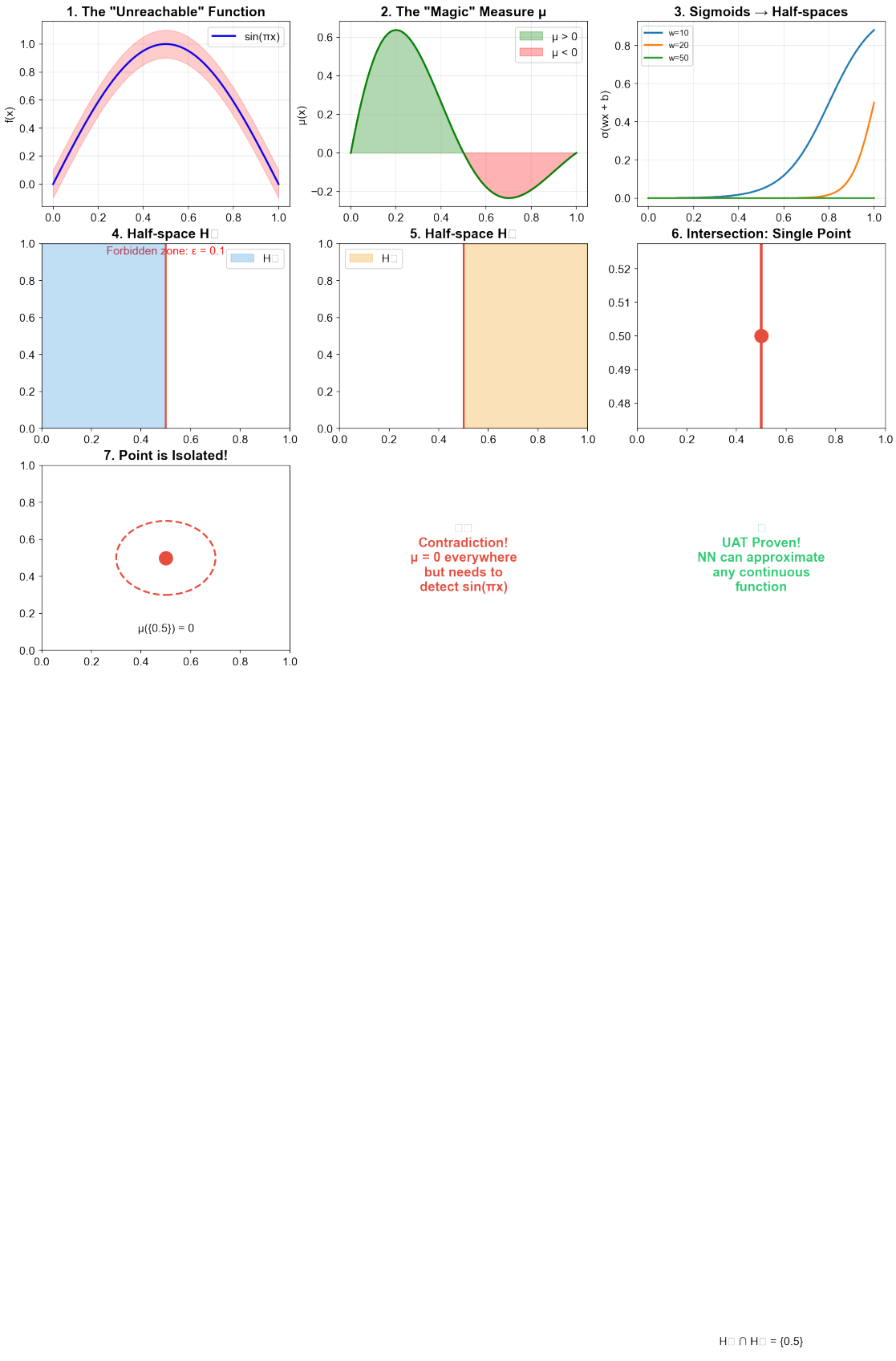

Figure 7: Proof by contradiction visualization. If neural networks cannot approximate a continuous function, a detector measure μ must exist. But half-spaces can isolate any point (panels 4-6), forcing $\mu = 0$ everywhere—a contradiction since μ must detect the target function.

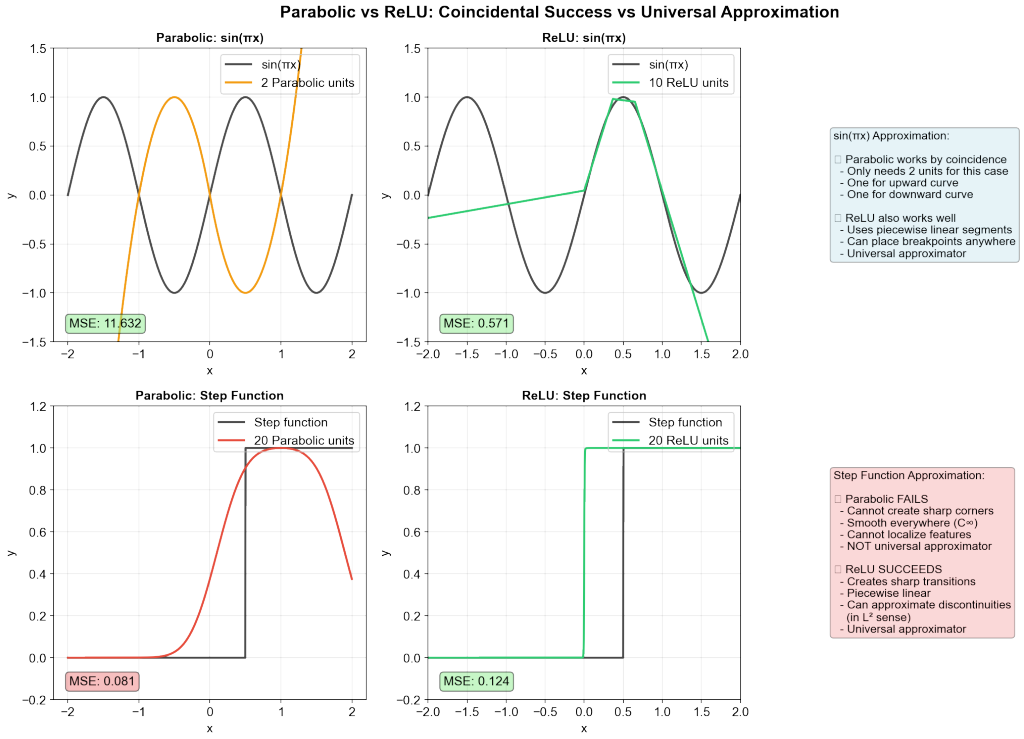

Figure 8: Parabolic activation's coincidental success vs general failure. While two parabolas can approximate $\sin(\pi x)$ (left), they fail for step functions (middle) because they cannot create localized features. The smooth, global nature of parabolic functions prevents universal approximation.

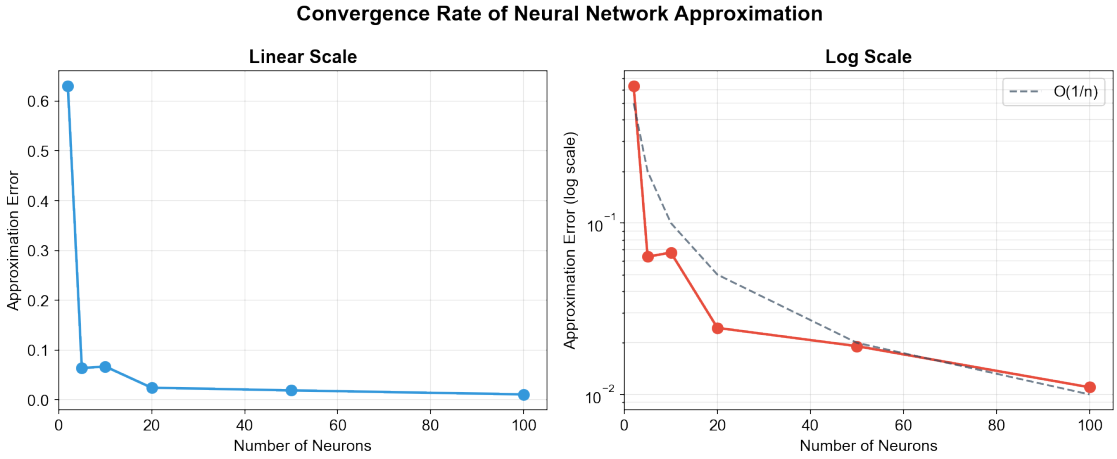

Figure 9: Approximation error decreases with network width. Linear scale (left) shows rapid initial improvement with diminishing returns. Log scale (right) reveals power law behavior: error $\sim N^{-\alpha}$ where N is the number of neurons.

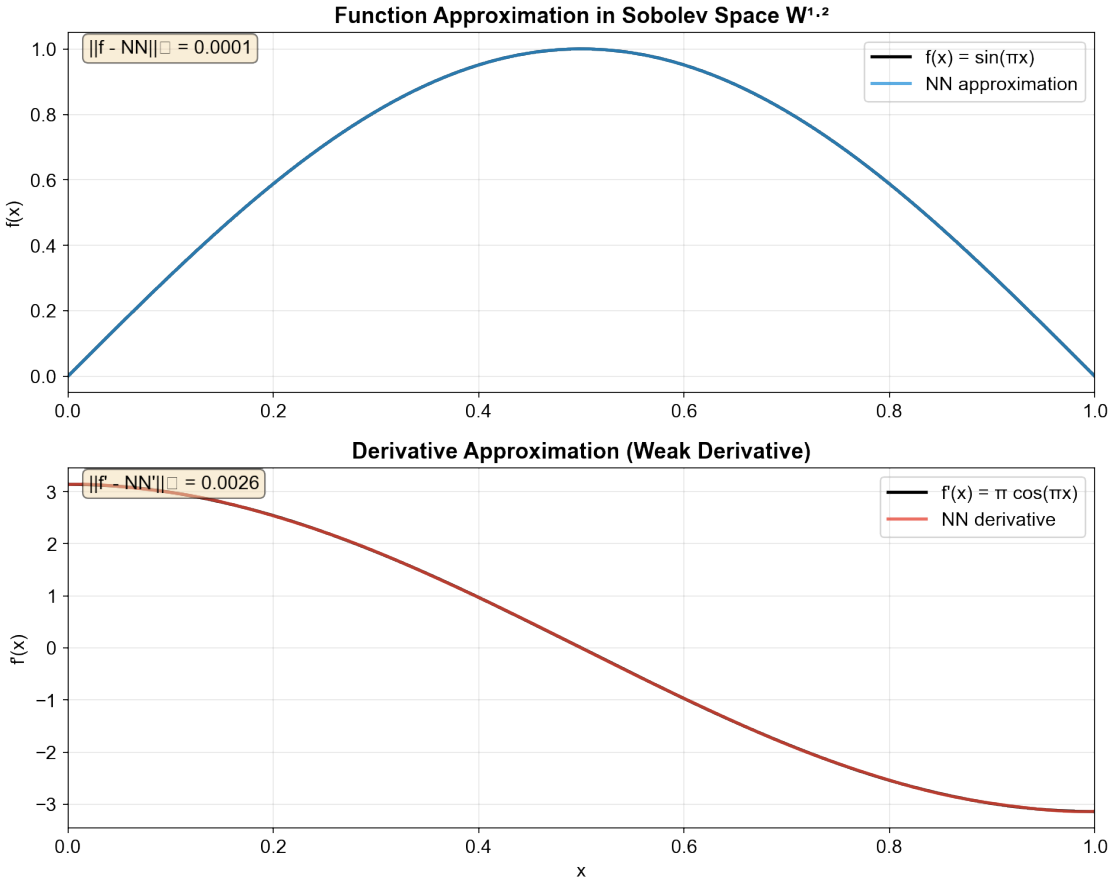

Figure 10: Neural networks approximate both function and derivative. Top: Function approximation of $\sin(\pi x)$ in L^2. Bottom: Derivative approximation showing larger errors but still convergent. This Sobolev space approximation is essential for solving PDEs.